Annotations of Classes and Inheritance Relationships: an Unified Mechanism in Order to Improve Skills of Library of Classes

Pierre Crescenzo, Christophe Jalady, Philippe Lahire

Annotations of Classes and Inheritance Relationships: an Unified Mechanism in Order to Improve Skills of Library of Classes

Pierre Crescenzo, Christophe Jalady and Philippe Lahire

Laboratoire I3S UNSA/CNRS - Projet OCL,
Les Algorithmes - Bât. Euclide - 2000, Route des Lucioles
BP 121 - F-06903 Sophia-Antipolis cedex - France
E-mail: {*Firstname.Lastname*}@unice.fr

Abstract

We propose here to show the usefulness of the addition of annotations inside the classes for a better specification of the use of the inheritance relationship in libraries of classes. The awaited added value is to improve the documentation, the reusability, the evolution capabilities and the robustness of these libraries; typically these annotations could be exploited by programming environments. We rely on existing taxonomies of inheritance to show the contribution of these annotations and we define an unified and flexible mechanism able to take into account any other taxonomy. We propose an integration of this mechanism in the Eiffel language through an extension of its clauses "inherit" and "indexing". The broad outline of the implementation based on model OFL defined in our team is also highlighted.

1. Introduction

Programming languages such as *Eiffel*, *Smalltalk*, *Java*, *C#* or *C++* provide a support for the specification of classes [1], inheritance relationship and aggregation (or client relationship) which have an expressiveness that may change from one language to the other, features (attributes, methods, etc.), visibility or protection rules that deal mainly with features but sometimes also with the classifiers themselves.

The expressiveness of those languages allows to take into account multiple situations but in some of them the designer does not have to express his choice explicitly or even it may not be supported. Although some improvement has been made in this direction - it is for example the case for the use of inheritance - it remains still much to do in this field. Thus, many papers discuss the justified uses of inheritance relationship [10, 11].

1. In the rest of the paper, we will prefer the more general term *classifier* used in UML[15].

For example in Java or Eiffel, the keyword *abstract* for the first and *deferred* for the second allows to stress that the classifier contains definitions of abstract or delayed routines, according to the vocabulary associated to each language. In Java, rather than to use only one inheritance relationship there are two keywords: *extends* which describes a single inheritance relationship between classes or a multiple one between interfaces, and *implements* which describes a multiple inheritance relationship between a class and one or several interfaces. Furthermore in C++ it is possible to inherit "privately" (keyword *private*), so that it is forbidden to apply polymorphism on class instances as it is normally the case when the symbol "::" is used.

According to our point of view these approaches represent a step forward because they allow the developer of application to better specify his use of inheritance but they seem still insufficient to us. Thus it can be interesting to give more explanation on the use of the keywords *extends* or *implements* in Java or of the keyword *inherit* in Eiffel, to quote only these two languages. In the same way researches like those of [9] or of [5] show that it is important to provide to a classifier a way to protect itself from the other entities of the program in order to avoid them to break the consistency of its instances. In our approach we wish to better take into account this aspect according to the use of inheritance relationships between classifiers.

Our proposal thus consists to allow the developer of application (in an optional way), to add a set of annotations to the description of the classifiers or to the inheritance relationship in order to better specify its use. It is important to mention that these annotations should be used by compilers, interpreters or programming environments rather than by the language runtime. They do not have vocation to modify the semantics of the inheritance relationships or more generally of the language. We promote the idea that thanks to this approach, a developer gets additional means in order to obtain more documented, more reusable, more maintai-

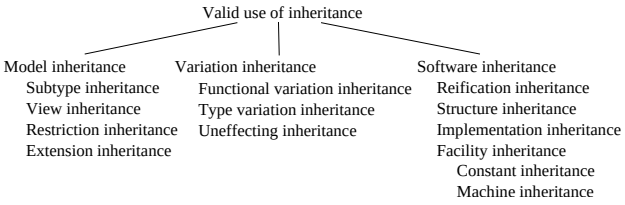

FIG. **1 –.** *Taxonomy proposed by B. Meyer*

nable and more robust applications.

We give initially an outline of the only two classifications dealing with inheritance relationship that we found in the state of the art and we propose a third classification (here known as *referential classificatio*n). Then we present the main elements which are useful for the definition of an annotation. In a third part we examine through an example the main aspects of our approach like its contribution in the development process of the libraries. In a fourth part we propose an integration of our approach in the Eiffel language [2]. We will finish by the evocation of related works and of mid-term perspectives for this work.

2. To classify information

In the state of the art we found only two taxonomies of inheritance. It is on the first hand the taxonomy of Bertrand Meyer [5]; its main goal is to clarify the various cases where the use of inheritance can be justified. On the other hand, the one proposed by Xavier Girod [7] aims to classify the various kinds of use of inheritance with the point of view of the application designer. We propose then a third classification which is orthogonal with the two first ones and whose objective is to make an inventory of the various limitations that one can apply to the nodes of these classifications without denaturing their meaning.

2.1. To justify the use of inheritance

We reproduce here the taxonomy suggested by Bertrand Meyer (see figure 1) and we briefly point out the meaning of the various types of inheritance by focusing especially on the most interesting aspects, i.e., the aspects which suggest the handling of some possible controls. It is useful to note that it applies more particularly to the Eiffel language and less to other languages, even if its matter can be generalized with profit.

– The **sub-type inheritance** allows to partition the set of instances of the ancestor class into disjoint sets represented by the subclasses. The super-class must be abstract and thus does not have proper instances.
– The **view inheritance** is used to operate a multi-criteria classification between the instances. It produces not disjoint partitions of instances and relies very much on multiple inheritance to represent the various “views” of an object. The classes concerned by this relationship have to be abstract.
– The **restriction inheritance** means that the subclass needs to satisfy an additional constraint compared to the ancestor class, such as for example a new invariant. The classes have to be or both abstract or both concrete.
– The **extension inheritance** allows to specify that the subclasses add characteristics which are not applicable to the super-class. The subclass which can be abstract or concrete extends the interface (i.e. the set of exported characteristics) of the class from which it inherits and that must be concrete.
– The **variation inheritance** deals with situations where the heir class wants to redefine one or more characteristics of the ancestor class. There is no addition of characteristics except possibly for the only need of redefined characteristics. One distinguishes the **functional variation** which allows the redefinition of the routine body, from the **type variation** which only aims to redefine its signature (type and numbers of arguments or type of the result). The subclass must be like its superclass abstract or concrete.
– The **uneffecting inheritance** leads to make abstract one or more characteristics of the ancestor class. It should be only used to merge two concrete routines having the same name in the case of multiple inheritance or to reuse a class which is too concrete and in this case, that represents a certain form of generalization. The source class [3] becomes naturally abstract.
– The **reification inheritance** is used when one wants to propose an implementation with initially abstract characteristics. Thus the ancestor class is inevitably deferred while the heir class can become concrete. In this type of inheritance the interface (the set of exported characteristics) of the ancestor class is not extended.
– The **structure inheritance** is used when the ancestor class represents a structural property which can be associated to the heir class, knowing that the structural property is only one aspect of the functionalities of the heir class. The ancestor class is inevitably abstract because the functionalities that it proposes are specific to the subclass.
– The **implementation inheritance** is used to provide to

2. Our approach could be easily adapted to other languages but integration would be surely different.

3. The source class is the one which defines the inheritance relationship and the target class is the inherited one.

the heir class some functionalities which will be used for the implementation of its behavior. This type of inheritance requires that ancestor and heir classes are both concretes. The latter can, moreover, define a second inheritance relationship enabling him to inherit from the model (it is a "*is a*"relationship) of this new super-class.

– The **facility inheritance** is mainly used to allow the heir classes to take advantage from some functionalities. For example, it will be used to inherit from a class which contains constants or once functions (**constant inheritance**) or to inherit from routines that may be viewed as operations on an abstract machine (**machine inheritance**).

2.2. To express the use of inheritance

In his classification, Xavier Girod [7] proposes seven uses of inheritance which are quite close from the classification of Bertrand Meyer. But two of them define a kind of "multiple" inheritance, that may encapsulate the other types of inheritance that have been highlighted. In the figure 2, these relationships are materialized by a vector (<h_1, ..., h_n>) where h_1, ..., h_n represent different types (respectively the same type), according to whether a **combined heritage** (respectively a **merge inheritance**), is considered.

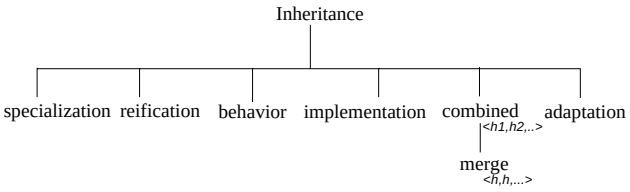

FIG. **2 –.** *Taxonomy proposed by Xavier Girod*

Thus the **combined inheritance** requiring the use of multiple inheritance, allows to specify a dependence between the highlighted inheritance relationships. For example a class *BOUNDED_STAC*K which inherits from *STACK* through a **reification inheritance** (close from the reification inheritance - see figure 1) and *ARRAY* thru an **implementation inheritance** (close from the implementation inheritance - see figure 1) implements indeed a combined inheritance: in this case the reification inheritance is associated to the implementation inheritance.

When the inheritance types are identical, and that the subclass represents an aggregate of ancestor classes in which no super-class has a dominating place; it is called **merge inheritance** (which can be seen like a specialization of the **combined inheritance**). An example of merge inheritance is the case of class *SEAPLANE* inheriting (specialization inheritance) from *BOAT* and *PLANE*.

In addition of these four inheritance relationships, X. Girod mentions also the **specialization inheritance**, the **behavior inheritance** and the **adaptation inheritance**, respectively close from the **extension inheritance**, the **structure inheritance** and the **variation inheritance** also proposed in the figure 1.

2.3. A classification known as of reference

We recalled the main aspects of two taxonomies on the usage of inheritance. We now propose a more technical classification in which we make an inventory of the elementary adaptations of the classifier behavior that it is possible to perform [4] on the source classifier when it inherits from it. This classification is orthogonal with the two preceding ones in the sense that it is not based on the same concerns. We will examine in the section 3 how it can contribute to improve the quality of libraries. It will be named in the rest of the paper the ***referential classification***.

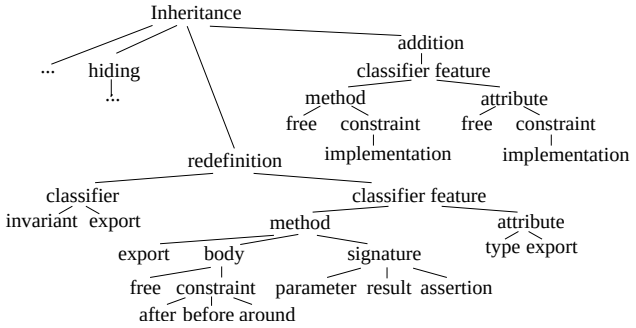

FIG. **3 –.** *Piece of the referential classification.*

The figure 3 represents only a piece of this classification, in which we are interested mainly in the adaptations dealing with the addition and the redefinition of characteristic. Among the categories of adaptation not quoted in the diagram there are for example, the renaming of characteristic or the uneffecting of a method body. In this classification, each node corresponds to a category of adaptation; it contains a property (noted *constraint*) which can take one of the three following values:

– **Mandatory**: the adaptation associated with the node must be applied for each instance of the corresponding meta-entity (in general a classifier, an attribute or a method). For example "a*ll the methods of the class must be redefined*".

– **Forbidden**: no occurrence of the corresponding meta-entity can carry out this adaptation.

4. We consider that we are in the same context as a developer who describes the source code of an application.

– **Allowed**: the adaptation associated with the node is possible, i.e. that it can or not, being applied to a given occurrence of the meta-entity.

For example it could be interesting to authorize (*allowed*) all the kinds of redefinition related to a method (*Inheritance/redefinition/classifier_feature/method*) or to prohibit (*forbidden*) the redefinition of the body of any method of the target classifier (*inheritance/...method/body/free*) or finally to make compulsory (*mandatory*) the redefinition of the body of any method of the classifier with the constraint that the new piece of code may only be added after the execution of the code of original method (*inheritance/.../body/constraint/after*). In the same way with regard to the addition of characteristics one will be able to allow (*allowed*), only the addition of method (*inheritance/addition/classifier_feature/method/free*) or even to allow (also *allowed*), only the addition of implementation method, i.e. methods not exported and used only by the body of the redefined methods (*inheritance/.../method/constraint/implementation*).

Moreover one or more rules is associated to each node of the classification. They define the semantics of the adaptation and depend naturally on the value associated with the property *constraint*.
sample of rule:

– node considered: *inheritance/...method/body/free*
– for each inheritance relationship *R*, *R.S* is its source classifier and *R.T* is its target classifier,
– $R.M_{ST}$ is the set of method of *R.S* which are redefined within *R.T*,
– *constraint = forbidden* **implies** $R.M_{ST}$ *= the empty set*

The suggested classification wants to be independent of the languages; it is thus probable that certain adaptations do not have any meaning in a given language. For example the redefinition of signature does not exist in Java whereas it is one of the facilities offered by the Eiffel language.

3. Towards the definition of an annotation

Our objective in this section is to show in an intuitive way how starting, from classifications of the state of the art dealing with the use of inheritance (figures 1 and 2), and also from the referential classification (figure3), it will be possible *i)* to give a definition of a classification node, *ii)* to highlight the fact that it is important to be able to customize the contents of these nodes and *iii)* to define what an annotation represents.

3.1. To define a node of taxonomy

We propose an approach for modeling and enrich information associated with an existing classification (mainly that of B. Meyer or X. Girod). Most of the time, information which describes the types of inheritance (nodes of the classification), is given in the form of a text or explanatory examples that may be understood differently according to who is reading. We propose at the same time, to extract from these comments, when it is possible, some formal rules which define their semantics, and to organize all information that may be catch in three categories [5]:

– **Information of adaptation**: they describe the adaptation capabilities of the source classifier and it can thus be described by a set of nodes of the classification of reference (it is called the **initial definition**). Of course this description is as much subjective as the explanations associated with classification are fuzzy or ambiguous and it depends on the expressiveness of the inheritance mechanism of the language. In the worst case one will associate the root of the classification of reference to a node.
– **Information on the structure of the hierarchy**: they depend on the total hierarchy. For example, in the case of a sub-typing relationship if B is a sub-type of A and C is also a sub-type of A, then the sets of the instances of A and B are disjoint. This information corresponds to constraints and they could be defined using meta-assertions
– **Intuitive information**: they correspond to textual descriptions or samples of the inheritance semantics. It is in this category that we will place information which could not be formalized and which are the most difficult to check.

In the rest of the paper we consider only the first two categories; third is out of the scope of this paper. The set of rules which will be listed on the level of a node N of taxonomy is equal to the union of the rules associated with its ancestors. These rules can be understood as being assertions at the meta-level of the application. It is pointed out that the set of rules which are part of the definition of adaptation information were defined in each node of the referential classification (see section 2.3).

3.2. To be able to customize a taxonomy

To know how to describe one node of a taxonomy like those of the figures 1 and 2 is not sufficient, it is necessary to be able to customize it in a specific way. This is due to the fact that one can be able, for some inheritance relationships taking part in the description of a library of classes, to specify the use of the inheritance relationship, more precisely than it is done in the initial definition of the node which is

5. It should be noted that any node of a taxonomy may refer to other nodes as it is the case in the figure 2.

considered. Our approach consists to allow the developer to modify, when it is necessary, the contents of the *information of adaptation* (i.e. the set of nodes which corresponds to the initial definition). We believe that the ability to put stronger constraints on the contents of a source classifier makes possible to increase at the same time the capabilities of control and automation of certain tasks related to the maintenance (or evolution) of the application. This contribution will be discussed again more in detail in the section 4.3, but let us consider right here possible samples of customization of the initial definition.

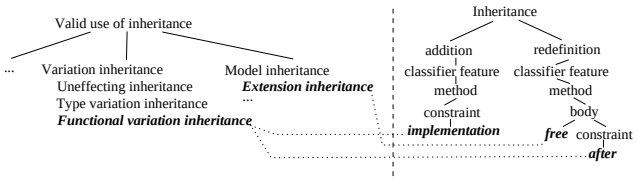

FIG. **4 –.** *Samples of customization*

In the figure 4 who contains extracts of the taxonomy of B. Meyer and of the referential classification we propose two examples of personalization. The first relates to the extension inheritance which allows the source classifier to freely redefine new methods, which was not authorized in the initial definition of this node.

The second relates to the functional variation inheritance. Here it is added two constraints to the source classifier which relate to the addition and the redefinition of characteristics; from now it is not possible:

- to redefine a method declared in the target classifier except if the redefinition only adds additional code after a call to the original method;
- to add methods except if they are called in the new body of the methods redefined in the source classifier and if they are not exported to classifiers other than itself;
- neither to redefine, nor to add other characteristics to the source classifier.

Of course they are only examples and other many interesting personalizations are possible for these two types of inheritance. Thus an alternative could be to authorize the free redefinition of the methods instead of constraining it. Another example not described in the figure 4, would be to consider the reification inheritance. It will be for example interesting to decline it either in a rather free form where the principal constraint would be to make compulsory the concretization of the set of features, or on the contrary in more strict forms which may rely on following assumptions:

- both source and target classifiers have the same interface (i.e. it is forbidden to add, modify or remove signatures) or,
- it is forbidden to redefine the concrete characteristics in the target classifier or,
- it is forbidden to modify the assertions or to add a clause to the invariant of the classifier.

Of course, other types of (compatible) constraint may be extracted from the classification of reference.

3.3. Definition of an annotation

Intuitively our needs according to an annotation are twofold. On the first hand we may associate a use *U* (for example *variation inheritance*) to an inheritance relationship *R* which could exist between two classifiers of a library. On the other hand we may record a possible redefinition of the initial definition of *U*. So that an annotation will contain following information:

- the name associated with a node of a taxonomy (for example *variation inheritance* if one considers the taxonomy of B. Meyer). It should be noted that this entity contains in particular a set (its initial definition), constituted of elements of the form:
  - a *name of node* of the classification of reference,
  - a *value* among *allowed*/*mandatory*/*forbidden*
- a set of nodes of the referential classification which represents, when it is needed, a redefinition of the initial definition of *U* (it is called **redefinition**).

The handling of the initial definition and of its redefinition will require to be equipped with set operators but also with operators making possible *i)* to add/remove elements to/from these sets, *ii)* to modify the value (*allowed*, *mandatory* or *forbidden*) associated to one of the elements of a set. We will detail a little more the implementation in the section 5.5, but one can already specify that the use of these annotations will lead to the definition of actions to be performed according to the result of the evaluation of the rules. These actions depend naturally on functionalities that one wants to introduce (for example in a programming environment), through the use of the annotations. Examples of functionalities are given in the section 4.

4. Possible contribution of the approach

We propose to study the contribution of the introduction of annotations into a library of data structures. We showed above the existence of two taxonomies, but others can also being considered; their various uses deal with the improvement of documentation, of reusability, of robustness, of the

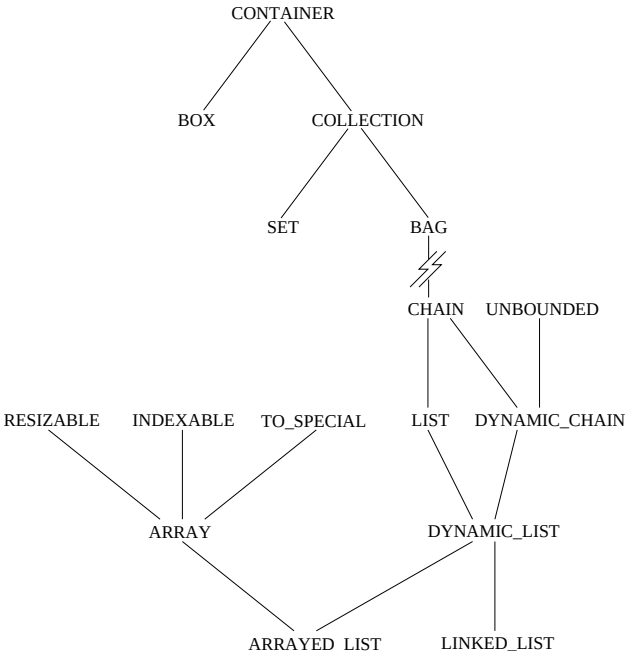

FIG. **5 –.** *A part of an Eiffel library of classes*

adaptability of a library of classes or finally of its design. We aim hereafter to address these various topics through the example described in figure 5.

The figure 5 represents a piece of the data-structure library of the Eiffel language [6]. We notice two classes at the bottom of the hierarchy, *ARRAYED_LIST* and *LINKED_LIST*, which implement in two different ways the concept of "traditional list" modeled by the class *DYNAMIC_LIST*. It will be noted that the latter is different from class *LIST* by the presence of characteristics such as one which make possible to add elements at the beginning and at the end of the list.

The super-hierarchy on the right-hand side of the figure represents the classification which allows to model the concept of *CONTAINER* (abstract data-structure representing a set of elements) and its various refinements such as *COLLECTION* (which includes features for adding or removing elements), *SET* (which guarantees the unicity of elements), *BAG* (which on the contrary leaves free the number of occurrences of an element), *CHAIN* (sequence, circular or not, of elements), etc.

Also on the figure, the left-hand side super-hierarchy specifies the concept of table. We included it mainly to show that it takes part in the implementation of class *ARRAYED_LIST*.

6. More precisely, this is the library which comes with EiffelStudio 5.3.

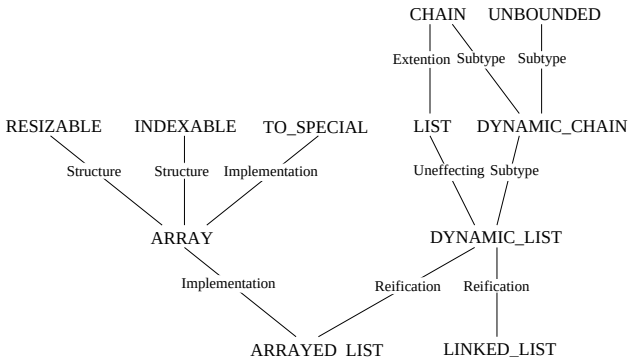

FIG. **6 –.** *Annotations on data-structure library*

4.1. Documentation and consistency

The figure 6 proposes a description of the use of inheritance according to the taxonomy suggested in the section 2.1. The *a posteriori* extraction of the use of inheritance, according to the information associated to the contents of the original library is a difficult work whose result is subject to polemic. It is thus likely to be called into doubt as we will show it later on. Our approach provides a formalism which allows the developer to annotate an inheritance relationship in order to describe its use according to this classification. It will be for him a way to justify the use of inheritance each time that it is needed. Such further thought is according to us essential in the process of both software design and implementation. It will be all the more useful as it is also possible to control that these annotations are consistent according to the implementation code.

To be able to justify to its project leader the use of inheritance compared to other solutions privileging the use of aggregation or client-supplier relationship is of course positive. But to offer also a first level of control (to ensure that the specified entities comply with some minimal rules associated with the various types of inheritance), is also important. To achieve such control we rely on the evaluation of the rules which describe the initial definition of each type of inheritance that is used. It will be interesting to try to recognize the types of inheritance in the main libraries and to check that they are consistent according to the source code; one will be able for example to generate checking-reports highlighting the possible inconsistencies.
The figure 7 allows to look at a piece of the data-structure library under a different point of view: the one expressed by the taxonomy of X. Girod (section 2.2). Thus its use allows to better address the problems related to the design in particular by taking into account the dependencies between several inheritance relationships; it thus favors the implementation of reverse-engineering processes. In the figure it is mentioned in particular a use of the *combined inhe-*

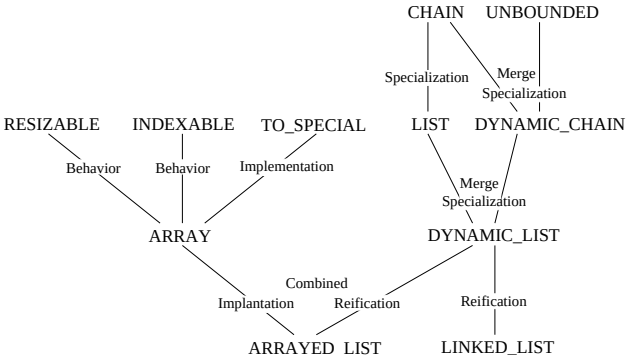

FIG. **7 –.** *Justification of the design choices*

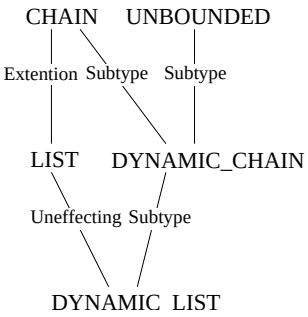

FIG. **8 –.** *Points of view on library of classes*

ritance and two of the *merge inheritance*. Thus the class *ARRAYED_LIST* implements the deferred methods of the class *DYNAMIC_LIST* with the ones defined in class *ARRAY*. In the same way the class *DYNAMIC_LIST* inherits in an equal way from the functionalities of the classes *LIST* and *DYNAMIC_CHAIN* and it customizes them when it is necessary.

4.2. Visualization of points of views

The figure 8 shows another benefit to annotate the inheritance relationship. Thus the annotations provide information which could be exploited to visualize various points of view of a library of classes and to present a less complex vision of the hierarchy. The interest is obvious: to allow the programmer to focus on the inheritance relationships it worries about in order for example to be convinced that the use of the sub-typing inheritance is consistent for the classes that are visualized in the figure. Another use of this information by programming environments can be to provide to a developer, which did not take part in the design of the library, the accurate vision in order to understand it gradually and calmly and to better consider its reuse or even its evolution.

4.3. Help for programming and debugging

This section is strongly based on the composition of classification presented in detail in section 2.3. In the development and debugging phase of an application it is important, for the developer, to take advantage from the maximum help of the programming environment. For example, according to the type of inheritance used within one class a specific text-editor could assist the programmer in its description. In the same way, still according to the type of inheritance, controls could be performed in order to inform the developer of possible errors. These controls will be as much relevant as what should be made in the descendant will be precisely described by the type of inheritance used. This is why we propose to redefine the initial definition associated with the nodes of taxonomy (figures 1 and 2). To be able to adapt the semantics of the inheritance use, probably by putting more constraints on it (and in any case by making sure that it keeps its relevance), allows to consider the automation of some activities of maintenance and also some assistances dealing with edition and debugging purposes. According to that, one will be able to add, exchange or remove elements which takes part in its initial definition. This is what is called in section 3.3 the *redefinition*.

4.4. To guarantee a better robustness

Another important aspect that it is possible to take into account with annotations is related to the protection of the classes with respect to those which may become their descendants (heirs). Although this aspect of the description of a program is associated to the use of "modifiers" most of the time, it can be interesting to annotate a class so that it specifies a little better what is likely to break its consistency if of adventure the heir classes use it badly. These annotations would be seen as information which could be used for example to generate a report which would synthesize the potential errors of design/programming and the potential risks that they could imply for the robustness of the application which is considered. Among the constraints that a class can wish to see applied by descendants one can quote:

- the use by the heir classes (when they inherit from it) of only some dedicated types of inheritance,
- limitation of the classifier that inherits from it to correspond to some specific type of classifier (interface, class, etc).

5. Integration of annotations for Eiffel

The choice to rather propose here an integration in the Eiffel language than in another one like Java for example,

may be explained by several observations:

– Java proposes several inheritance relationships identified explicitly by various keywords (*extends*, *implements*) or implicitly by the type of target classifier (*interface*, *class*). Eiffel proposes a single mechanism;
– Both classification found in the state of the art rely partially on facilities provided only by Eiffel (multiple inheritance, covariance, advanced mechanism of assertion, etc.).

5.1. Annotation of inheritance: basic mechanism

The Eiffel language proposes already an *indexing* clause described mainly before the description of a class; it allows to give additional information on the class which is intended to be used by external tools or programming environments. This information is described using several independent lines that contain *i)* a name indicating the type of information, *ii)* the character ":" and *iii)* one string which corresponds either to a sentence for comment purpose, or a sequence of words separated by commas.

example:

```
description: "this class ..."
keywords: "word1, word2, ..."
...
```

The list of the types of information (here *description* and *keywords*) is free and can be extended according to the needs of the developer. It is particularly important to ensure a perfect uniformisation of the keywords that are used at least inside the same library, because it makes possible then to consider implementation of systematic searches of information. Our approach integrates the concept of annotation through the extension of the inheritance clause (*inherit*) with an indexing clause (*indexing*). The declaration below defines an example of use. One can note that the choice to add this information in the clause *indexing* rather than through other syntactic additions insists on the fact that the semantics of the language is not modified and that information will be used by external tools. Just as for the clause *indexing* associated to a class, the clause *indexing* of inheritance is optional. In this example, the clause *indexing* specifies that the classification used (*taxonomy*) is the one described in the figure 1 and that the node chosen within the classification represents a reification inheritance (*use*).

```
class LINKED_LIST [G]
inherit
 DYNAMIC_LIST [G]
  rename
    ....
  redefine
    ....
  indexing
    taxonomy: "valid_use_inherit"
    use: "reification_inheritance"
  end
...
end -- class
```

5.2. To constraint inheritance annotations

The example below is dealing with the same situation as previous one but it wishes to indicate in this class several specific limitations through the use of the contents of the classification of reference (figure 3). To avoid making the class less readable by using the full path of the nodes as it is made in the section 2.3 we preferred to give here a name which wants to be as significant as possible. It should be noted that the operators **+**, **-** and **=** respectively mean to add, remove and replace an element of the initial definition. In the diagram below, the programmer means through their evocation that it is possible *i)* to redefine in a constrained way the body of the routines (it was free before, and he must contain from now, before the added code, the call to the original version of the method), *ii)* to add attributes or implementation methods which should not be exported to other classes, and *iii)* to make effective all the methods (before it was optional). The letters **A**, **F** and **M** correspond respectively to *allowed*, *forbidden* and *mandatory* (see section 2.3).

```
indexing
   ...
  taxonomy_inherit: "valid_use_inherit"
class LINKED_LIST [G]
inherit
 DYNAMIC_LIST [G]
 ....
 indexing
  use: "reification_inheritance"
  redefinition: "
       - method_redefinition_body
       + method_redefinition_after (A),
       + add_implementation_feature (A),
       = method_body_definition (M)"
 end
...
end -- class
```

It will be noted that if the *indexing* clause associated to each inheritance relationship which is annotated relates to the same classification, it could be interesting to factorize this information in the *indexing* clause of the class in order not to repeat several times the same information.

5.3. To allow the use of combined inheritance

Our mechanism fits also to the use of other classifications as the one described in the figure 2 which wants to express

that several inheritance relationships have to coexist in order to express a need. This is why one specifies a different name for *taxonomy_inherit*: it identifies the taxonomy of X. Girod.

```
indexing
   ...
 taxonomy_inherit: "use_of_inherit"
class ARRAYED_LIST [G]
inherit
  DYNAMIC_LIST [G]
  ....
   indexing
    use: "realization"
    encompassed_type: "combined_inherit 1"
  end
  ARRAY [G]
   ....
   indexing
    use: "implementation"
    encompassed_type: "combined_inherit 1"
  end
...
end -- class
```

In the example above it is specified the type of inheritance associated to the reuse of classes *DYNAMIC_LIST* and *ARRAY* which are respectively reification and implementation inheritances. However it is also stated that the inheritance of these two classes takes part in the implementation of a more complex inheritance use which is called *combined_inheritance*. The added number (here *1*) is optional. It is only used to avoid any ambiguity in the event of the use of several combined inheritances.

5.4. Annotation of classifiers

It can be interesting to specify in a class that its possible use by another class through an inheritance relationship, must satisfy a set of constraints. For example, as it is proposed below, a class agrees to be a target only when the corresponding inheritance relationship is associated to a precise use (*inherit_restrict*).

```
indexing
   ...
 inherit_restrict: "type_variation_inherit,
                    functional_variation_inherit"
class ARRAYED_LIST [G]
inherit
   DYNAMIC_LIST [G]
   ARRAY [G]
...
end -- class
```

Class *ARRAYED_LIST* allows only variation inheritance (either according to the type or to the functional aspects). the reader will note that to force the descendants of a class to check certain properties, does not imply to annotate the inheritance in the class itself; however in order to be able to always provide a better control, we recommend it.

5.5. Implementation of the annotation mechanism

To implement our approach requires to be able to describe all characteristics dealing with the mechanism of annotations, especially the rules and the constraints which allow to describe each node of the referential classification. These rules rely on the reification of an application such as for example the methods or the attributes of a class, the list of the redefined methods, etc. We want to carry out this implementation starting from the OFL/J API which offers a full object reification [7] of the set of entities of an application, and is equipped with functionalities to integrate new meta-entities. The extension of model OFL defined in [13] makes it possible to define meta-assertions that will rely on the reification proposed by the model. The reification of an annotation and of all elements necessary for the description of a taxonomy will be added to the OFL/J library. For more simplicity for the one who is in charge of the description of classifications, the set of the meta-assertions and rules could be described in OCL (Object Constraint Language - UML) [15], and then transformed in order to be integrated into the reification of the application.

The generation of the set of objects that participate to the reification could be carried out by an external tool included in the programming environment or the compiler. Naturally for efficiency reasons we will reify only information strictly necessary to the exploitation of the annotations. Some of the actions to be performed were highlighted in the section 4 but the list is not exhaustive (generation of report, display following various points of views, consistency checks, etc.), could be integrated by the programming environment by using approaches by Separation of Concerns or simply through the design pattern *visitor* [14].

6. Related work

First of all [8] proposes thanks to the use of a meta-object protocol (MOP), to specify the nature of the classes, i.e. to specify some specific properties like to be abstract or not to be able to have subclasses. The paper deals with *ClassTalk* which is based on the SmallTalk language with the addition of a MOP. Other characteristics are introduced like the possibility to specify the set of methods to be redefined by the subclasses or to forbid the modification of the interface. All these characteristics are encapsulated in meta-classes; it is the role of each class to specify the meta-class it is an instance of. These characteristics should influence the possible future inheritance relationships between a class (instance of one given meta-class) and of its (future) subclasses. In a certain manner, the paper proposes to allow a class to put constraints on the future inheritance relationships that may

7. This library is a first implementation of meta-model OFL and for the moment it is written in Java.

declare it as target. It concentrates on the improvement of the structural organization of the classes but not on properties making possible to consider additional controls on the hierarchies. However, by increasing the understanding of a component, it proposes a use for the improvement of the programming environments.

All work relating to the “reverse-engineering” is concerned with problems of maintenance and evolution of hierarchies of classes. Research work of P. Clarke and B Malloy [12] presents a taxonomy of class in order to better describe the modification made in various versions of a component. Thus they propose a set of properties which characterize the classes, like to be generic, to be abstract, to represent a thread, or to characterize its methods: to trigger an exception, or to declare protections. This taxonomy will allow to instantiate a model on various versions of the components closely connected to the change that are carried out. Among other related works we should mention research around the design models like UML (Unified Modeling Language of the OMG) and more particularly the UML profiles [15].

7. Conclusion and perspectives

We proposed a flexible approach to annotate the classifiers and more precisely the inheritance between classifiers. We showed that to write a classifier by annotating it requires a further thought and a greater rigor. However we are persuaded that the benefit which it is possible to get when these annotations are taken into account by the programming environments is the improvement, of the reuse, the adaptability, the documentation and the robustness of the libraries of classes. The risk is that by restricting the objective of a classifier as the use of the annotations seems to encourage it, we note a proliferation of classifiers. To avoid this drawback, it will be necessary to use the annotations only when that brings something significant. Some points are still to improve; they deal in particular with the description and the handling of classification for which we must better take into account the state of the art. The same applies to the study of problems connected to reverse-engineering. Finally it could be interesting to extend our approach in such manner that it is also possible to define the possible uses of classifiers.

References

[1] *T. Lawson, C. Hollinshead, and M. Qutaishat, "The potential for Reverse Type Inheritance in Eiffel",* Technology of Object-Oriented Languages and Systems TOOLS13, *Prentice Hall, 1994, pp. 349-357.*

[2] *P. Crescenzo, and P. Lahire, "Using both Specialisation and Generalisation in a Programming Language: Why an How?",* Advances in Object-Oriented Information Systems OOIS 2002 Workshops, *Montpellier, September 2002, pp. 64-73.*

[3] *M. Sakkinen, "Exheritance, Class Generalization Revived",* Object Oriented Programming ECOOP-2002 The Inheritance Workshop, *June 2002.*

[4] *P. Crescenzo, OFL: Un modèle pour paramétrer la sémantique opérationelle des langages à objets - applicaton aux relations inter-classes, PhD. Thesis, University of Nice-Sophia Antipolis, December 2001.*

[5] *B. Meyer,* Object-Oriented Software Construction 2nd edition, *Prentice-Hall, 1997.*

[6] *G.L. Steele,* Common Lisp the Language 2nd edition, *Digital Press, 1990.*

[7] *X. Girod, Conception par objets - MECANO: une méthode et un environnemnet de construction d’application par objets, PhD. Thesis, University of Joseph Fourier Grenoble I, Grenoble, June 1991.*

[8] *T. Ledoux, and P. Cointe, "Les métaclasses explicites comme outil pour améliorer la conception des bibliothèques de classes", GDR’95, Grenoble, 1995.*

[9] *G. Ardourel, Modélisation des mécanismes de protection dans les langages à objets, PhD Thesis, University of Montpellier II, Montpellier, December 2002.*

[10] *A. Taivalsaari, "On the Notion of Inheritance",* ACM Computing Surveys, *ACM press, September 1996, Vol. 28 No. 3 pp. 438-479.*

[11] *D.C Halbert, and P.D O’Brien, "Using Types and Inheritance in Object-Oriented Languages",* Object Oriented Programming ECOOP-1987, *1987, pp. 20-31.*

[12] *P. Clarcke, B. Malloy, and P. Gibson, "Using a Taxonomy Tool to Identify Changes in Object-Oriented Software", to appear in* Conference on Software Maintenance and Re-engineering 2003, *2003.*

[13] *D. Pescaru, and P. Lahire, “Modifiers in OFL - An Approach For Access Control Customization”, I3S Laboratory Research Report, May 2003, pp. 10.*

[14] *E. Gamma, and all, “Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley Professional Computing Series, *April 1996.*

[15] *Object Management Group, “Unified Modeling Language Specification(UML), Version 1.5, Mars 2003.*